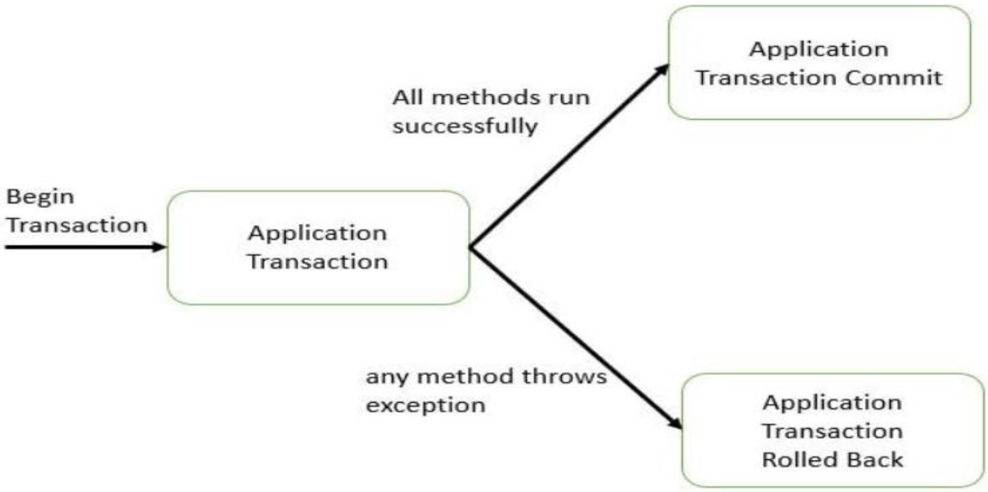
1. **Explain Spring Transaction Management?**

Spring provides powerful **transaction management** features that ensure data consistency and integrity in **multi-step operations** within applications. It handles transactions declaratively or programmatically, ensuring **ACID (Atomicity, Consistency, Isolation, Durability)** compliance in database operations.

Let's understand transactions with the above example. If a user has entered his information, the user's information gets stored in the user_info table. Now, to book a ticket, he makes an online payment, and due to some reason(system failure), the payment has been canceled, so the ticket is not booked for him. But, the problem is that his information gets stored on the user_info table. On a large scale, more than thousands of these things happen within a single day. So, it is not good practice to store a single action of the transaction(Here, only user info is stored, not the payment info).
To overcome these problems, Spring provides transaction management, which uses annotation to handle these issues. In such a scenario, the spring stores the user information in temporary memory and then checks for payment information. If the payment is successful, then it will complete the transaction; otherwise, it will roll back the transaction, and the user information will not be stored in the database.



2. **How to handle the Spring Transaction Management?**

@Transactional Annotation

In Spring Boot, @Transactional annotation is used to manage transactions in a Spring Boot application and to define a scope of transaction. This annotation can be applied to the class level or method level. It provides data reliability and consistency. When a method is indicated with @Transactional annotation, it indicates that the particular method should be executed within the context of that transaction. If the transaction becomes successful, then the changes made to the database are committed; if any transaction fails, all the changes made to that particular transaction can be rollback and it will ensure that the database remains in a consistent state.

> ***Note:*** *To use @Transactional annotation, you need to configure transaction management. However, if you are using spring-boot-starter-data-jpa, Spring Boot auto-configures transaction management, and you do not need to explicitly add @EnableTransactionManagement. If you are not using spring-boot-starter-data-jpa or need custom transaction management, you can enable it by adding @EnableTransactionManagement to your main class of the Spring Boot application.*

3. **Explain different types of Transaction management?**

Types of Spring Transactions

1. **Declarative Transaction Management (Recommended)**
   - Uses `@Transactional` annotation on **service-layer methods**.
   - No need for explicit `beginTransaction()` or `commit()` calls.
2. **Programmatic Transaction Management**
   - Uses `TransactionTemplate` or `PlatformTransactionManager` to **manually** control transactions.
   - Offers flexibility for **custom rollback** and commit handling.

Example of Declarative Transaction Management

Java

```java
import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Service;

@Service
public class BankService {

    @Transactional
    public void transferMoney(Account from, Account to, double amount) {
        from.withdraw(amount);
        to.deposit(amount);
    }
}
```

✅ If an error occurs during withdrawal or deposit, **Spring will automatically roll back the transaction.**

Example of Programmatic Transaction Management

```java
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class TransactionalService {
    private PlatformTransactionManager transactionManager;

    public TransactionalService(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void processTransaction() {
        TransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);

        try {
            // Perform database operations
            transactionManager.commit(status);
        } catch (Exception e) {
            transactionManager.rollback(status);
        }
    }
}
```

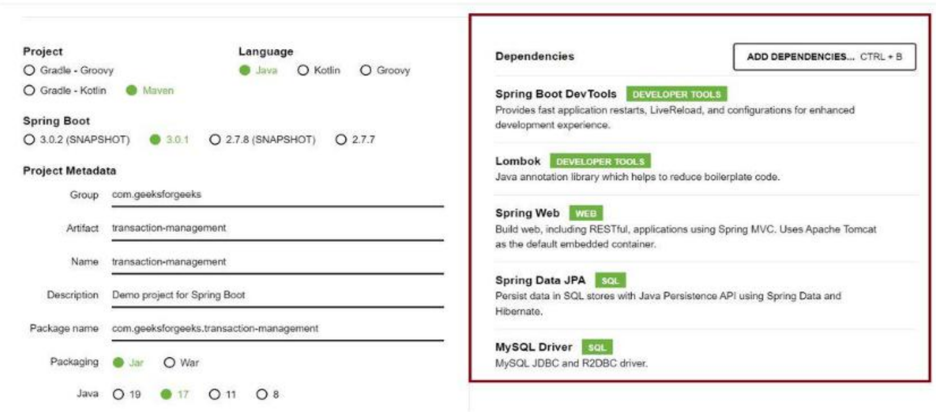
4. Give an example for Spring Boot Declarative transaction management?

Step 1: Create A Spring Boot Project

In this step, we will create a spring boot project. For this, we will be using Spring Initializr. To create a spring boot project please refer to How to Create a Spring Boot Project?

Step 2: Add Dependencies

We will add the required dependencies for our spring boot application.



Step 3: Configure Database

Now, we will configure the database in our application. We will be using the following configurations and add them to our **application.properties** file.

```
server.port = 9090
#database configuration
spring.datasource.url=jdbc:mysql://localhost:3306/employee_db
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57Dialect
#the ddl-auto=update : It will create the entity schema and map it to db automatically
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Note: *Please add your database username & password along with the database path.*

Step 4: Create Model Class

In this step, we will create our model class. Here, we will be creating two model classes, **Employee** and **Address**. While creating the model class we will be using [Lombok Library](#).

Employee.java

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Entity
@Table(name="EMP_INFO")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;

}
```

Address.java

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Entity
@Table(name="ADD_INFO")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String address;

    // one to one mapping means,
    // one employee stays at one address only
    @OneToOne
    private Employee employee;

    @Version
    private Long version;

}
```

Step 5: Create a Database Layer

In this step, we will create a database layer. For this, we will be creating **EmployeeRepository** and **AddressRepository** and will be extending **JpaRepository<T, ID>** for performing database-related queries.

EmployeeRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;
import com.geeksforgeeks.transactionmanagement.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

AddressRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;
import com.geeksforgeeks.transactionmanagement.model.Address;

public interface AddressRepository extends JpaRepository<Address, Integer> {

}
```

Step 6: Create a Service Layer

You can use **@Transactional** annotation in service layer which will result interacting with the database. In this step, we will create a service layer for our application and add business logic to it. For this, we will be creating two classes **EmployeeService** and **AddressService**. In EmployeeService class we are throwing an exception.

EmployeeService.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.geeksforgeeks.transactionmanagement.model.Address;
import com.geeksforgeeks.transactionmanagement.model.Employee;
import com.geeksforgeeks.transactionmanagement.repository.EmployeeRepository;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private AddressService addressService;

    @Transactional
    public Employee addEmployee(Employee employee) throws Exception {
        Employee employeeSavedToDB = this.employeeRepository.save(employee);

        Address address = new Address();
        address.setId(123L);
        address.setAddress("Varanasi");
        address.setEmployee(employee);

        // calling addAddress() method
        // of AddressService class
        this.addressService.addAddress(address);
        return employeeSavedToDB;
    }
}
```

AddressService.java

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.geeksforgeeks.transactionmanagement.model.Address;
import com.geeksforgeeks.transactionmanagement.repository.AddressRepository;

@Service
public class AddressService {

    @Autowired
    private AddressRepository addressRepository;

    public Address addAddress(Address address) {
        Address addressSavedToDB = this.addressRepository.save(address);
        return addressSavedToDB;
    }

}
```

Step 7: Create Controller

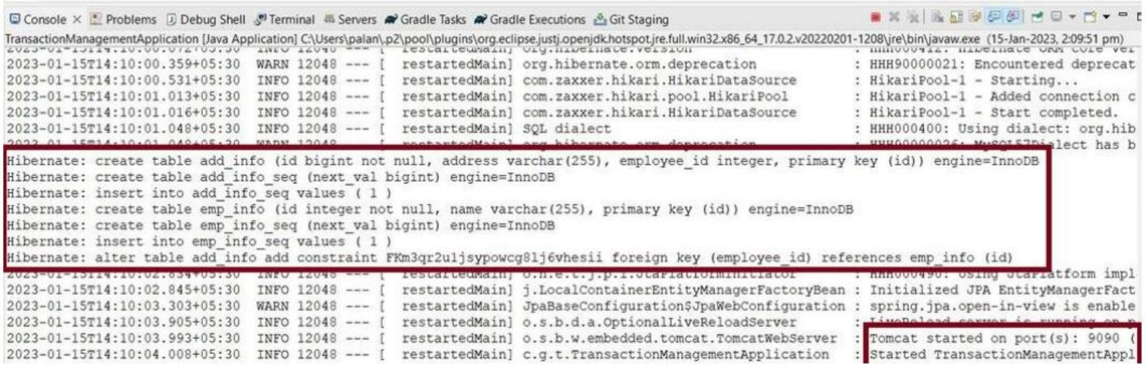
In this step, we will create a controller for our application. For this, we will create a Controller class and add all the mappings to it.

Controller.java

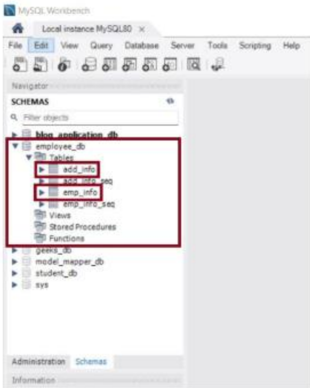
```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.geeksforgeeks.transactionmanagement.model.Employee;
import com.geeksforgeeks.transactionmanagement.service.EmployeeService;

@RestController
@RequestMapping("/api/employee")
public class Controller {

    @Autowired
    private EmployeeService employeeService;

    @PostMapping("/add")
    public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee) throws Exception{
        Employee employeeSavedToDB = this.employeeService.addEmployee(employee);
        return new ResponseEntity<Employee>(employeeSavedToDB, HttpStatus.CREATED);
    }
}
```

Step 8: Running Our Application

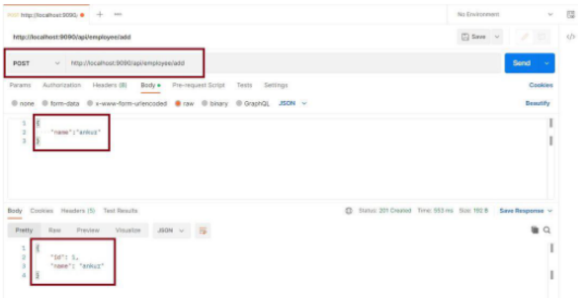
In this step, we will run our application. Once, we run our application using hibernate mapping in our database required tables will be created.



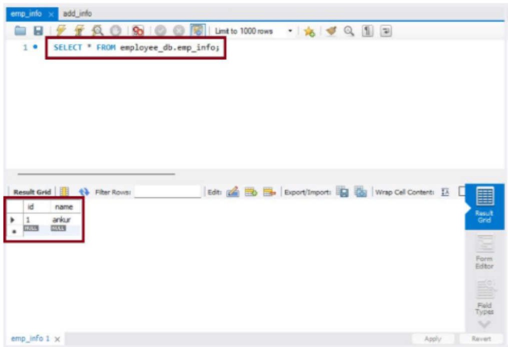
As we can see in logs, our table has been created. We can also confirm it by looking at our database.



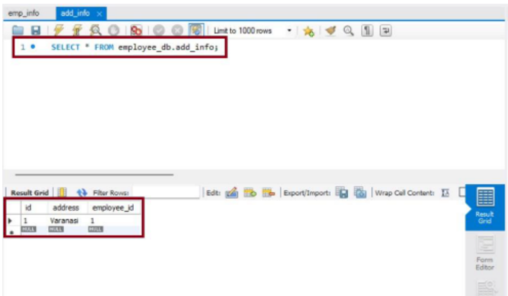
Now, we will request our application for adding an employee to it, using postman. To learn more about postman please refer to [Postman – Working, HTTP Request & Responses](). Once, we hit the request, **the request moves from the controller to the service layer where our business logic is present**.



As we can see in the above response we have added an employee. We can also check our database for employee data and address data.



Similarly, we can also check for address data.



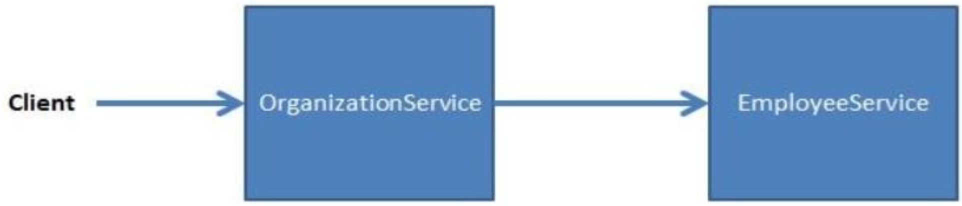
5. Explain different types of propagation in transaction management?

Transaction propagation determines how transactions interact when multiple transactional methods are executed. Spring provides different **propagation types** to define how transactions behave within nested calls.

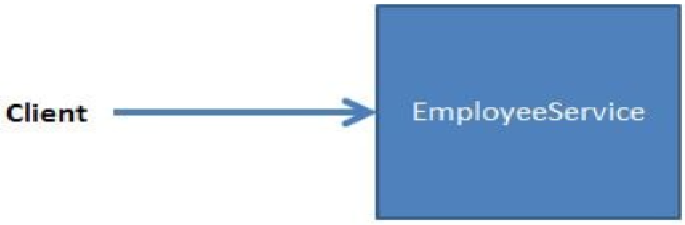
Propagation	Behaviour
REQUIRED	**Always executes in a transaction.** If there is any existing transaction it uses it. If none exists then only a new one is created
SUPPORTS	**It may or may not run in a transaction.** If current transaction exists then it is supported. If none exists then gets executed with out transaction.
NOT_SUPPORTED	**Always executes without a transaction.** If there is any existing transaction it gets suspended
REQUIRES_NEW	**Always executes in a new transaction.** If there is any existing transaction it gets suspended
NEVER	**Always executes with out any transaction.** It throws an exception if there is an existing transaction
MANDATORY	**Always executes in a transaction.** If there is any existing transaction it is used. If there is no existing transaction it will throw an exception.

But suppose the user wants to call the Employee Service in both ways i.e.

- Call using Organization service



- Call the the Employee Service directly.

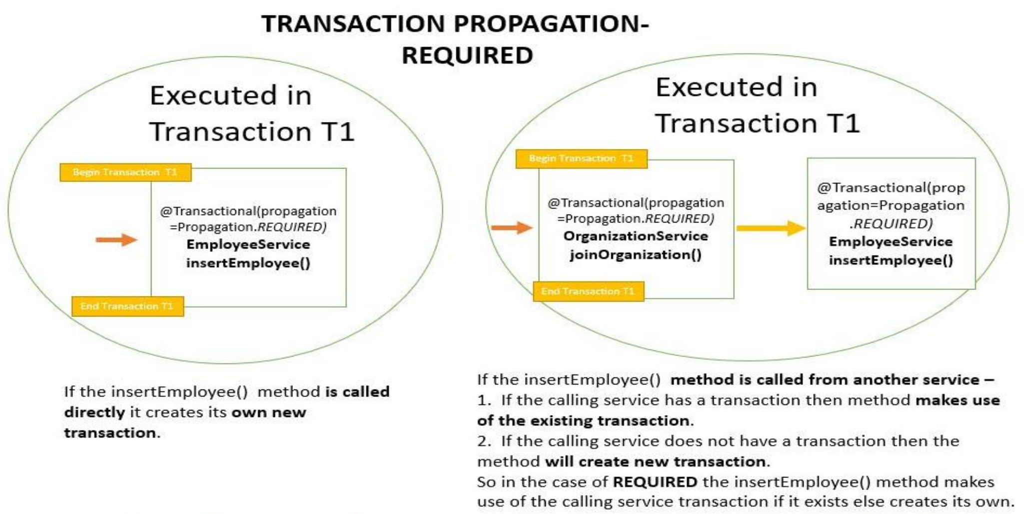


As the Employee Service may also be called directly we will need to use Transaction annotation with Employee Service also. So both the services - Organization Service and the Employee Service will be using Transaction annotation.

We will be looking at the various propagation scenarios by observing the behaviour of the Organization and Employee service. There are six types of Transaction Propagations-

- **REQUIRED**
- **SUPPORTS**
- **NOT_SUPPORTED**
- **REQUIRES_NEW**
- **NEVER**
- **MANDATORY**

Transaction Propagation - **REQUIRED** (Default Transaction Propagation)

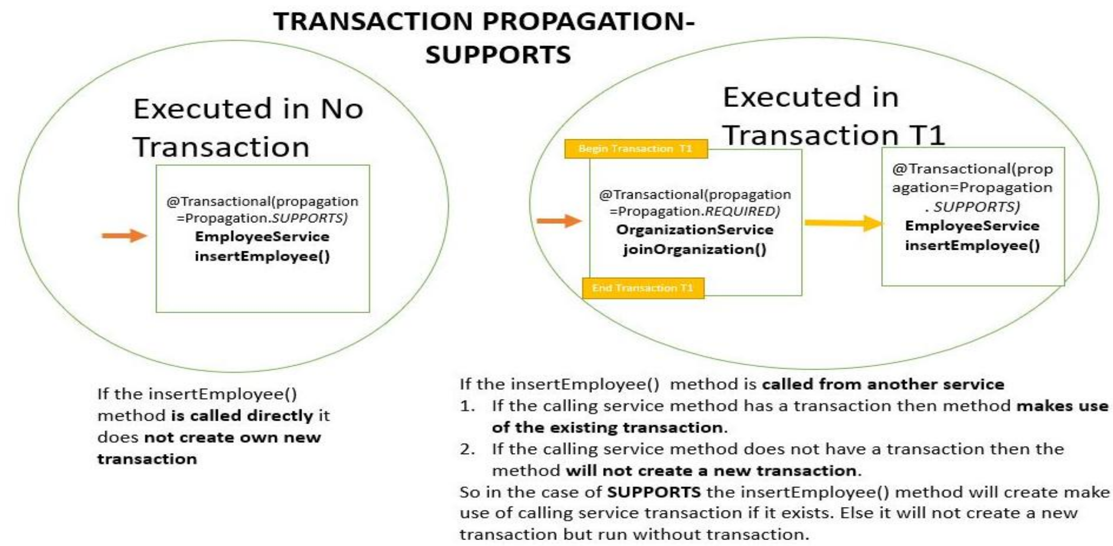


Here both the Organization Service and the Employee Service have the transaction propagation defined as **Required**. This is the default Transaction Propagation.

Example

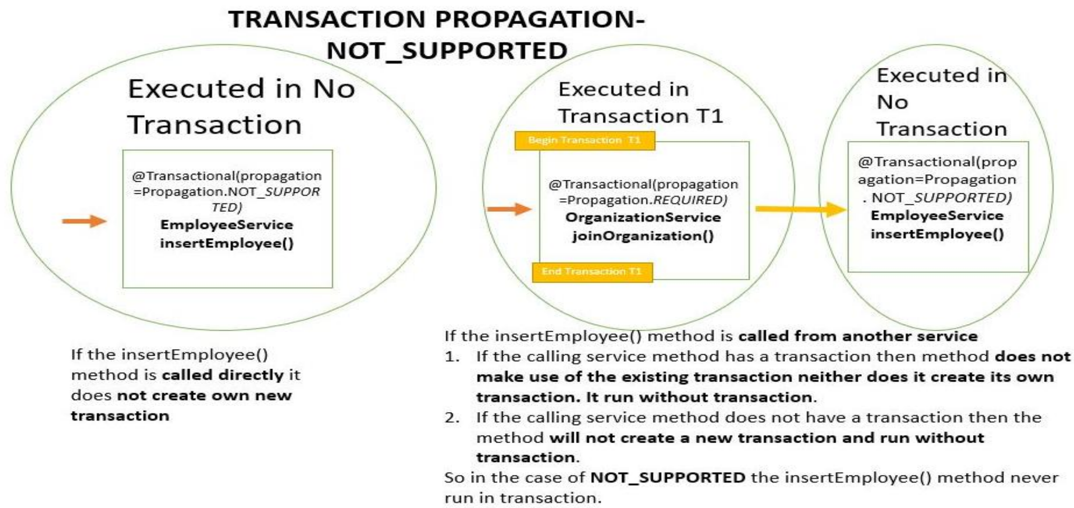
```
@Transactional(propagation = Propagation.REQUIRED)
public void requiredExample(String user) {
    // ...
}
```

Transaction Propagation - **SUPPORTS**



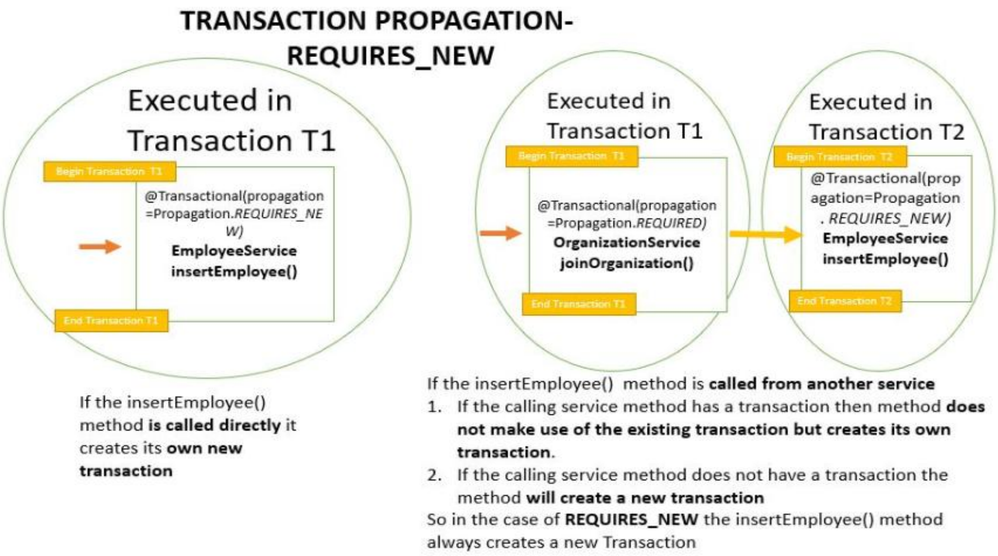
Here both the Organization Service has the transaction propagation defined as **Required** while Employee Service the transaction propagation is defined as **Supports**.

Transaction Propagation - **NOT_SUPPORTED**



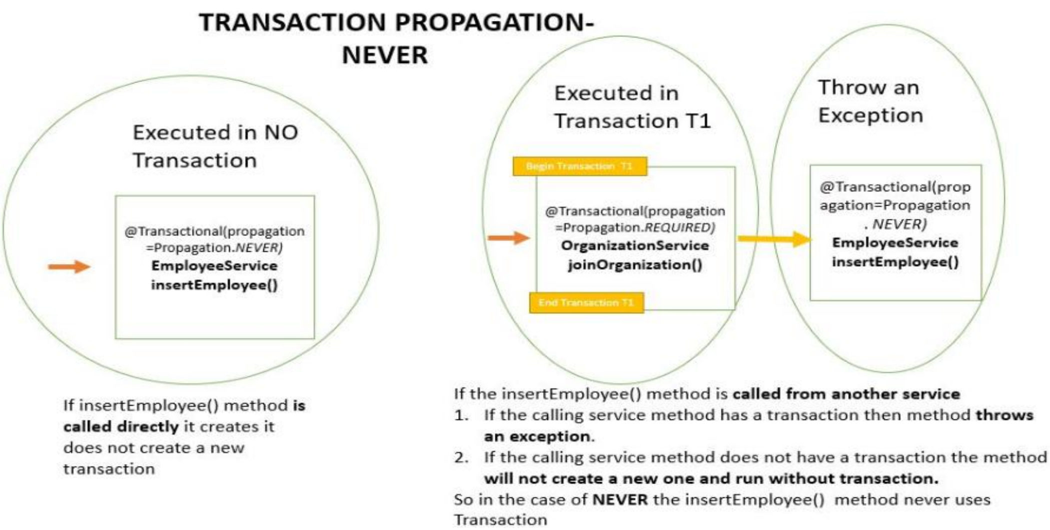
Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **NOT_SUPPORTED**

Transaction Propagation - **REQUIRES_NEW**



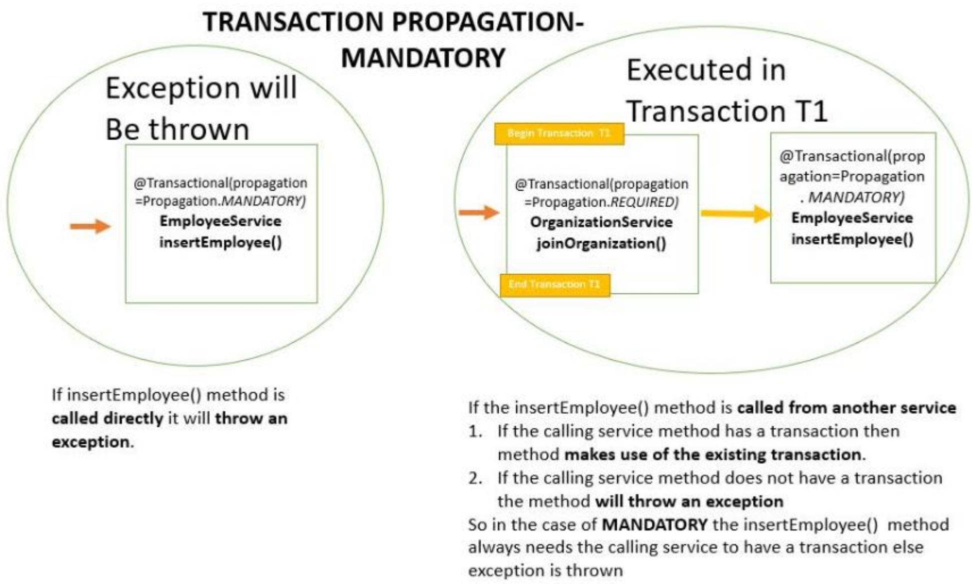
Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **REQUIRES_NEW**

Transaction Propagation - **NEVER**



Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **NEVER**s

Transaction Propagation - **MANDATORY**



Here for the Organization Service we have defined the transaction propagation as **REQUIRED** and the Employee Service have the transaction propagation defined as **MANDATORY**

6. **Explain about different types of Transaction Isolation?**

Transaction Isolation controls **how and when** the changes made by one transaction become visible to other concurrent transactions. It's a key part of maintaining **data consistency**.

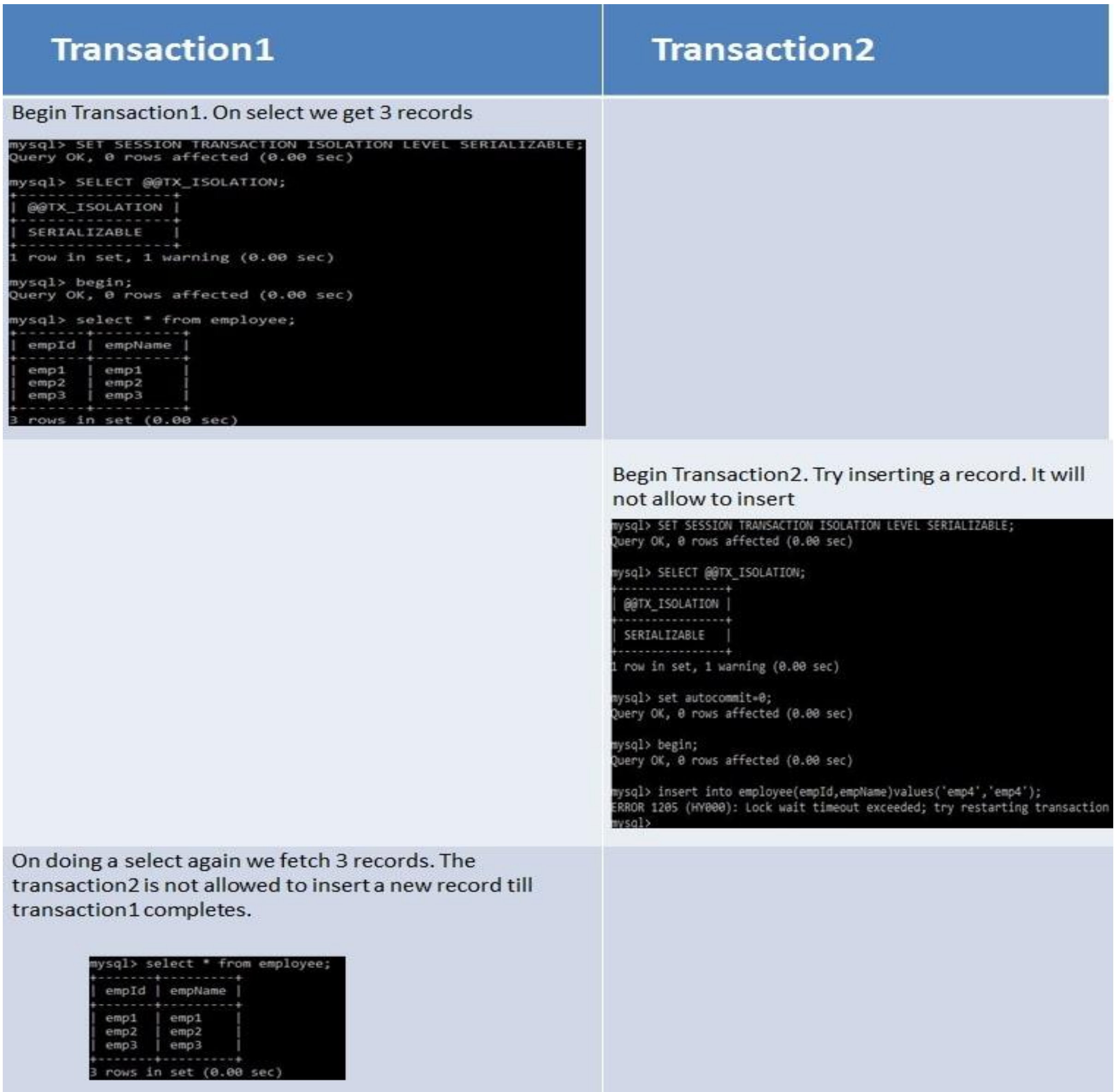
Types of Transaction Isolation Levels

Spring supports the following **isolation levels**, based on database configurations:

Isolation Level	Description
READ UNCOMMITTED	Transactions see **uncommitted changes** of other transactions, leading to **dirty reads**.
READ COMMITTED	Transactions see **only committed changes**, preventing **dirty reads** but allowing **non-repeatable reads**.
REPEATABLE READ	Ensures consistent reads, preventing **dirty and non-repeatable reads**, but not **phantom reads**.
SERIALIZABLE	The strictest isolation, completely **avoiding concurrency issues** but reducing performance.

SERIALIZABLE

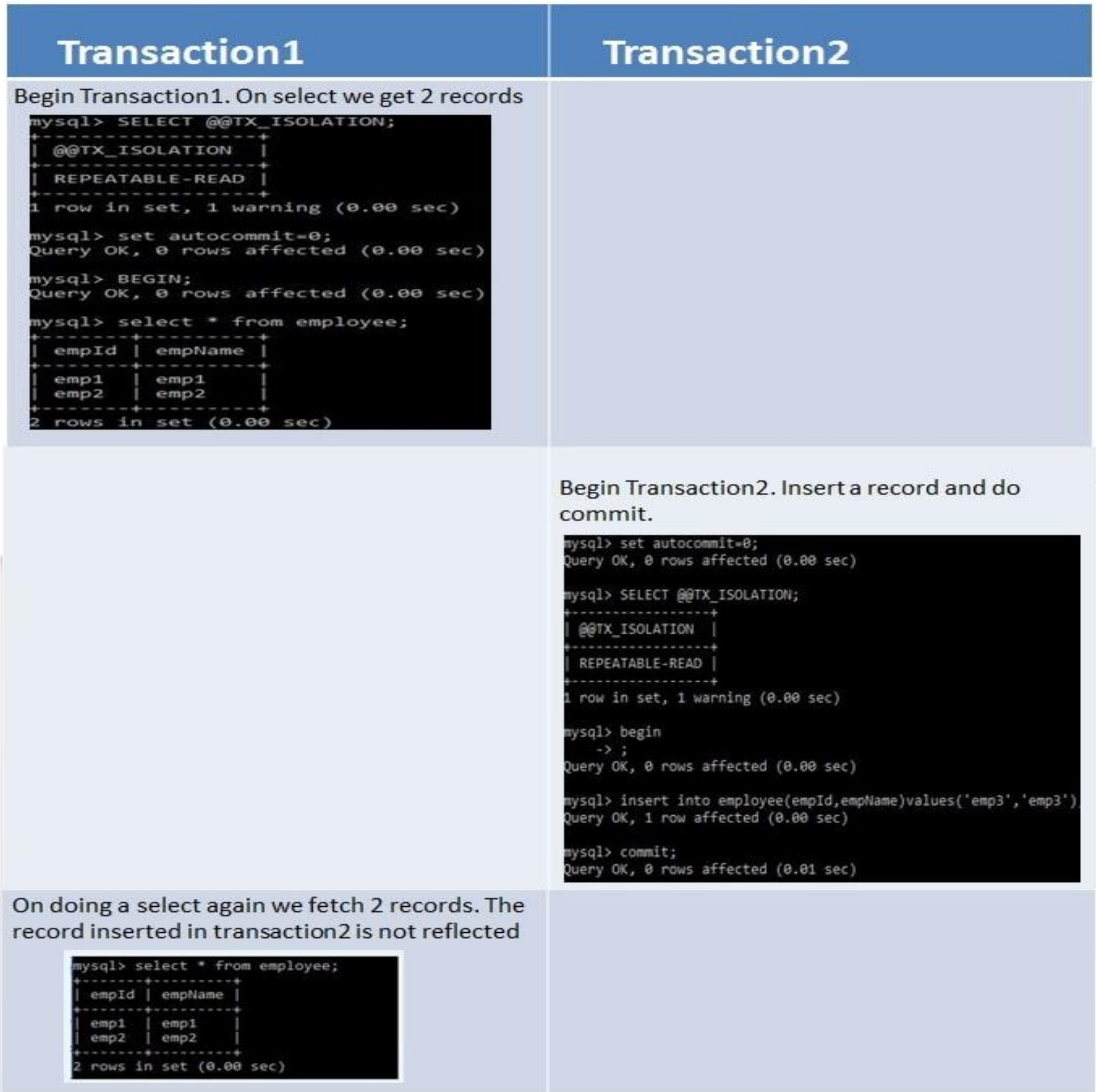
If two transactions are executing concurrently then it is as if the transactions get executed serially i.e the first transaction gets committed only then the second transaction gets executed. This is **total isolation**. So a running transaction is never affected by other transactions. However this may cause issues as **performance will be low and deadlock might occur**.



REPEATABLE_READ

If two transactions are executing concurrently - **till the first transaction is committed the existing records cannot be changed by second transaction but new records can be added.** After the second transaction is committed, the new added records get reflected in first transaction which is still not committed. For MySQL the default isolation level is REPEATABLE_READ.

However the REPEATABLE READ isolation level behaves differently when using mysql. When using MYSQL we are not able to see the newly added records that are committed by the second transaction.



READ_COMMITTED

If two transactions are executing concurrently - **before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction.** After the second transaction is committed, the newly added and also updated records get reflected in first transaction which is still not committed.

<table>
<tr><th>Transaction1</th><th>Transaction2</th></tr>
<tr><td>Begin Transaction1. On select we get 4 records
<pre>
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
emp1	emp1
emp2	emp2
emp3	emp3
emp4	emp4
+-------+---------+
4 rows in set (0.00 sec)
</pre></td><td></td></tr>
<tr><td></td><td>Begin Transaction2. Insert a record and do commit.
<pre>
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into employee(empId,empName)values('emp5','emp5');
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)
</pre></td></tr>
<tr><td>On doing a select again we fetch 5 records. The record inserted in transaction2 is reflected in transaction1 only after transaction 2 is committed.
<pre>
mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
emp1	emp1
emp2	emp2
emp3	emp3
emp4	emp4
emp5	emp5
+-------+---------+
5 rows in set (0.00 sec)
</pre></td><td></td></tr>
</table>

READ_UNCOMMITTED

If two transactions are executing concurrently - before the first transaction is committed the existing records can be changed as well as new records can be changed by second transaction. **Even if the second transaction is not committed the newly added and also updated records get reflected** in first transaction which is still not committed.

Transaction1	Transaction2
Begin Transaction1. On select we get 3 records	
	Begin Transaction2. Insert a record and do not do a commit.
On doing a select again we fetch 4 records. The record inserted in transaction2 is not committed, but still gets reflected in transaction1.	

Transaction1 — Begin Transaction1. On select we get 3 records

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
| emp3  | emp3    |
+-------+---------+
3 rows in set (0.00 sec)
```

Transaction2 — Begin Transaction2. Insert a record and do not do a commit.

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@TX_ISOLATION;
+------------------+
| @@TX_ISOLATION   |
+------------------+
| READ-UNCOMMITTED |
+------------------+
1 row in set, 1 warning (0.00 sec)

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into employee(empId,empName)values('emp4','emp4');
Query OK, 1 row affected (0.00 sec)
```

Transaction1 — On doing a select again we fetch 4 records. The record inserted in transaction2 is not committed, but still gets reflected in transaction1.

```
mysql> select * from employee;
+-------+---------+
| empId | empName |
+-------+---------+
| emp1  | emp1    |
| emp2  | emp2    |
| emp3  | emp3    |
| emp4  | emp4    |
+-------+---------+
4 rows in set (0.00 sec)
```

Summary

- **Dirty Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A modifies a record but not commits it. Transaction B reads this record but then Transaction A again rollbacks the changes for the record and commits it. So Transaction B has a wrong value.
- **Non-Repeatable Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B modifies these records before transaction A has been committed. So if Transaction A again reads these records they will be different. So same select statements result in different existing records.
- **Phantom Reads -** Suppose two transactions - Transaction A and Transaction B are running concurrently. If Transaction A reads some records. Transaction B adds more such records before transaction A has been committed. So if Transaction A again reads there will be more records than the previous select statement. So same select statements result in different number records to be displayed as new records also get added.

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
SERIALIZABLE	This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially	This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially	This scenario is not possible as the second transaction cannot start execution until the first is committed. They never execute parallelly but only sequentially
REPEATABLE_READ	This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value.	This scenario is not possible since any record can be changed only after a transaction has been committed. So multiple select statements before transaction commit will always return same existing records.	This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place.
READ_COMMITTED	This scenario is not possible as any existing record change gets reflected only if the transaction is committed. So other transaction will never read wrong value.	This scenario is possible as other transactions can modify existing records even if first transaction commit has not taken place.	This scenario is possible as other transactions can insert new records even if first transaction commit has not taken place.
READ_UNCOMMITTED	This scenario is possible as any record can be read by other transactions even if the first transaction is not committed. So if first transaction rollbacks the record changes then other transactions will have wrong values	This scenario is possible since any record can be changed even if a transaction is not committed.	This scenario is possible as any record can be inserted even if a transaction is not committed.

Example:

```
@Service
public class OrganzationServiceImpl implements OrganizationService {

	@Autowired
	EmployeeService employeeService;

	@Autowired
	HealthInsuranceService healthInsuranceService;

	@Override
	// Using Transactional annotation we can define any isolation level supported by the underlying database.
	@Transactional(isolation = Isolation.SERIALIZABLE)
	public void joinOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {
		employeeService.insertEmployee(employee);
		healthInsuranceService.registerEmployeeHealthInsurance(employeeHealthInsurance);
	}

	@Override
	@Transactional
	public void leaveOrganization(Employee employee, EmployeeHealthInsurance employeeHealthInsurance) {
		employeeService.deleteEmployeeById(employee.getEmpId());
		healthInsuranceService.deleteEmployeeHealthInsuranceById(employeeHealthInsurance.getEmpId());
	}
}
```

7. Explain Rollback Mechanism in Spring Boot Transaction Management?

Rollback ensures **data integrity** by undoing changes **when an error occurs** during a transaction. Spring Boot **automatically handles rollback** in transactions based on **exception types and configurations**

How Rollback Works in Spring Boot

✅ If a transaction **fails**, Spring **reverts all database changes** made within that transaction.

✅ By default, **unchecked exceptions (** `RuntimeException` **,** `Error` **) trigger rollback**.

✅ **Checked exceptions (** `Exception` **) do NOT trigger rollback automatically**, but can be explicitly configured.

Example: Automatic Rollback on RuntimeException

```java
import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Service;

@Service
public class BankService {

    @Transactional
    public void transferMoney(Account from, Account to, double amount) {
        from.withdraw(amount);
        to.deposit(amount);

        if (amount > 10000) { // Simulating an error
            throw new RuntimeException("Transaction amount exceeds limit!");
        }
    }
}
```

Rollback for Checked Exceptions

To roll back **checked exceptions**, explicitly set `rollbackFor` in `@Transactional`:

Java Copy

```java
@Transactional(rollbackFor = Exception.class)
public void processOrder(Order order) throws Exception {
    order.reserveStock();
    order.makePayment();

    if (!order.isValid()) {
        throw new Exception("Invalid order!");
    }
}
```

🚀 **Without** `rollbackFor` **, Spring will NOT roll back for checked exceptions.**

Handling Rollback Programmatically

```java
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class PaymentService {
    private PlatformTransactionManager transactionManager;

    public PaymentService(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void processTransaction() {
        TransactionStatus status = transactionManager.getTransaction(new DefaultTransactionDefinition());

        try {
            // Perform database operations
            transactionManager.commit(status);
        } catch (Exception e) {
            transactionManager.rollback(status); // Manual rollback
        }
    }
}
```

8. Can we apply @Transactional in class level and method level?

Yes, **@Transactional** can be applied at both the **class level** and **method level** in Spring.

1️⃣ Class-Level `@Transactional`

✅ When applied at the **class level**, it **affects all methods** inside the class **unless a method has its own transactional configuration**.
✅ Provides a **default transaction behavior** for the entire class.
✅ Individual methods **can override** the class-level transaction settings.

Example: Applying `@Transactional` at Class Level

Java

```java
import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Service;

@Service
@Transactional // Applied to all methods in this class
public class AccountService {

    public void deposit(Account account, double amount) {
        account.addBalance(amount);
    }

    public void withdraw(Account account, double amount) {
        account.reduceBalance(amount);
    }
}
```

🚀 **All methods (`deposit`, `withdraw`) execute within a transaction automatically.**

2️⃣ Method-Level `@Transactional`

✅ When applied at the **method level**, it affects **only that specific method**, overriding any class-level transactions.
✅ Useful for **customizing individual methods**, e.g., **different propagation types or rollback settings**.

Example: Overriding `@Transactional` at Method Level

```
@Service
@Transactional // Default transaction applies to all methods
public class OrderService {

    public void placeOrder(Order order) {
        order.process();
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW, rollbackFor = Exception.class)
    public void cancelOrder(Order order) throws Exception {
        order.reverseTransaction();
        throw new Exception("Order cancellation failed!");
    }
}
```

🚀 **The `cancelOrder()` method runs in a separate transaction and rolls back on all exceptions.**

Key Differences Between Class-Level & Method-Level `@Transactional`

Aspect	Class-Level `@Transactional`	Method-Level `@Transactional`
Scope	Applies to **all methods** in the class	Applies **only to the specific method**
Override	Can be overridden by method-level transaction	Overrides class-level configuration
Flexibility	Less customizable for individual methods	Allows **specific configurations** per method

9. How to handle distributed transaction management in spring boot micro services?

```
Handling ACID (Atomicity, Consistency, Isolation, Durability) in distributed transaction management is complex because it involves multiple systems or services potentially across
different networks or databases. Here's how it's generally done:

1. Two-Phase Commit (2PC)
✅ Ensures atomicity by coordinating transactions across multiple databases.
✅ Uses a coordinator to prepare and commit transactions in two steps.
🚨 Downside: Can cause performance bottlenecks and blocking issues.

🔹 2. Saga Pattern
✅ Breaks a transaction into multiple compensatable steps across microservices.
✅ If a step fails, a compensating transaction rolls back previous changes.
🚨 Downside: Requires careful design to handle failures properly.

🔹 3. Eventual Consistency with Event-Driven Architecture
✅ Uses asynchronous events to propagate changes across services.
✅ Ensures consistency over time rather than immediate atomicity.
🚨 Downside: Requires monitoring and reconciliation mechanisms.
```

10. Explain distributed transaction management in micro services?

🚀 Distributed Transaction Management in Microservices

In a **monolithic application**, managing a transaction is simple because all operations occur within the same database and JVM.
But in **microservices**, each service typically manages **its own database** — this makes **distributed transaction management** challenging.

🧩 What is a Distributed Transaction?

A **distributed transaction** spans **multiple services and databases**, and all operations must either **commit or rollback** as a single unit.

😰 Challenges

- Each microservice is **independently deployed**, often using different databases.
- There's **no shared transaction context** across services.
- Network failures and partial failures are common.

2. **Saga Pattern** – *Asynchronous, Eventual Consistency* ✅

Instead of a global transaction, a **Saga** is a sequence of **local transactions**, where each service:

- Performs a transaction
- Publishes an event
- The next service listens and acts
- If something fails, **compensating transactions** are triggered

```
------------------------------------------------------------
Here's a simple Choreography Pattern example in Spring Boot using Kafka. The Choreography pattern is a decentralized communication style often used in microservices, where services
react to events rather than being orchestrated by a central service.

🔧 Scenario: Order Processing System
We have 3 microservices:

Order Service - Creates an order and publishes an event.

Payment Service - Listens to the order-created event, processes payment, and publishes a payment-processed event.

Inventory Service - Listens to the payment-processed event and updates stock.

🧱 Technology Stack
Spring Boot

Spring Kafka

Apache Kafka

📦 Kafka Topics
order-created
payment-processed

1. Order Service
Producer: Order Created Event

@Service
public class OrderService {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void createOrder(String orderId) {
        // Business logic to create order...
        kafkaTemplate.send("order-created", orderId);
    }
}
```

```
2. Payment Service
Consumer: Order Created Event

@Service
public class PaymentService {

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    @KafkaListener(topics = "order-created", groupId = "payment-group")
    public void processPayment(String orderId) {
        // Business logic to process payment...
        kafkaTemplate.send("payment-processed", orderId);
    }
}

3. Inventory Service
Consumer: Payment Processed Event

@Service
public class InventoryService {

    @KafkaListener(topics = "payment-processed", groupId = "inventory-group")
    public void updateInventory(String orderId) {
        // Business logic to update inventory...
        System.out.println("Inventory updated for order: " + orderId);
    }
}
```

```
📘 application.yml (for all services)

spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: <appropriate-group-id>
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer

🧠 Notes
Each service is independent and communicates via Kafka topics.

No centralized coordinator (orchestrator); services react to events.

Use schemas (e.g., Avro, Protobuf, or JSON) for structured message exchange in production.
```

11. Explain ACID properties?

Property	Description
Atomicity	All or nothing — either the entire transaction succeeds, or none of it does
Consistency	Ensures data moves from one valid state to another
Isolation	Transactions don't interfere with each other
Durability	Once committed, the transaction is permanent

1️⃣ Atomicity (All or Nothing)

Ensures that **a transaction is either fully completed or fully rolled back**. If any step fails, **changes made so far are undone**.

Example: Bank Money Transfer

```java
import org.springframework.transaction.annotation.Transactional;
import org.springframework.stereotype.Service;

@Service
public class BankService {

    @Transactional
    public void transferMoney(Account from, Account to, double amount) {
        from.withdraw(amount);
        to.deposit(amount);

        if (amount > 10000) {
            throw new RuntimeException("Transaction exceeds limit!");
        }
    }
}
```

2️⃣ Consistency (Valid State Before & After Transactions)

Ensures that **database integrity constraints** are maintained before and after the transaction.

Example: Enforcing Data Integrity

Java Copy

```java
@Transactional
public void createOrder(Order order) {
    if (!order.hasValidCustomer()) {
        throw new RuntimeException("Invalid customer ID!");
    }
    order.save();
}
```

🚀 Prevents saving orders that **violate integrity rules**, ensuring database consistency.

3️⃣ Isolation (Concurrency Control)

Defines **how transactions interact concurrently**, preventing data corruption.

Example: Setting Transaction Isolation Level

Java

```java
@Transactional(isolation = Isolation.READ_COMMITTED)
public void checkBalance(Account account) {
    account.fetchBalance();
}
```

🚀 **Prevents dirty reads** by ensuring changes from other transactions are visible **only after commit**.

4️⃣ Durability (Ensures Persistence)

Guarantees that **once a transaction is committed, changes are permanent**.

Example: Writing to a Transaction Log

Java

```java
@Transactional
public void saveTransactionLog(TransactionLog log) {
    log.writeToDatabase();
}
```

🚀 Even if the system crashes, committed transactions are **recoverable through transaction logs**.

12. What happens if a transaction times out in Spring?

✅ Spring **automatically rolls back** the transaction if it exceeds the configured timeout.

✅ Use `timeout` parameter in `@Transactional` to specify the max execution time.

💡 **Example: Setting Transaction Timeout**

Java

```java
@Transactional(timeout = 5) // Transaction times out after 5 seconds
public void longRunningProcess() {
    // Heavy database operations
}
```

🚀 **Ensures slow transactions do not block database resources.**

13. **What is the benefit of using @Transactional(readOnly = true)?**

✅ Improves **performance** by avoiding locks on the database.
✅ Optimizes **read operations**, reducing transaction overhead.

💡 **Example: Using `readOnly = true` for performance optimization**

```java
@Transactional(readOnly = true)
public List<Customer> fetchCustomers() {
    return customerRepository.findAll();
}
```

🚀 **Ideal for queries where no data modification occurs.**

14. **What happens if you call a transactional method inside another transactional method?**

✅ If the called method **uses `REQUIRED` propagation**, it runs within the parent transaction.

✅ If the called method **uses `REQUIRES_NEW` propagation**, it starts a new transaction **independent of the parent**.

✅ If the parent transaction **rolls back**, transactions using `REQUIRED` also rollback, but transactions using `REQUIRES_NEW` remain committed.

💡 **Example: Nested Transactions**

```java
@Transactional
public void transferMoney(Account from, Account to, double amount) {
    from.withdraw(amount);
    to.deposit(amount);

    logService.logTransaction(from, to, amount); // Uses REQUIRES_NEW
}
```

🚀 **If transferMoney() fails, balance changes roll back, but logging remains committed.**

15. **How does Spring handle rollback in transactions?**

✅ By default, **unchecked exceptions (`RuntimeException`, `Error`) trigger rollback.**
✅ Checked exceptions (`Exception`) **do not trigger rollback**, unless explicitly set using `rollbackFor`.

💡 **Example: Rollback on Checked Exception**

```java
@Transactional(rollbackFor = Exception.class)
public void placeOrder(Order order) throws Exception {
    order.processPayment();
    order.updateStock();
    if (!order.isValid()) {
        throw new Exception("Invalid Order!");
    }
}
```

🚀 Without `rollbackFor = Exception.class`, **Spring will commit the transaction even if an exception occurs.**

16. What is the difference between Declarative and Programmatic Transaction Management?

Type	Declarative Transactions (`@Transactional`)	Programmatic Transactions (`PlatformTransactionManager`)
Implementation	Uses `@Transactional` annotation	Uses `TransactionTemplate` or `TransactionManager`
Control	Spring handles transactions automatically	Full control over transaction handling
Code Complexity	**Simpler** – No explicit commit/rollback code	**More complex** – Requires manual commit/rollback
Recommended For	Standard transactions (JPA, JDBC, Hibernate)	Advanced transaction handling (nested transactions, manual rollbacks)

💡 **Example: Programmatic Transaction Management**

```java
TransactionTemplate transactionTemplate = new TransactionTemplate(transact
transactionTemplate.execute(status -> {
    try {
        account.withdraw(100);
        account.deposit(100);
        return true;
    } catch (Exception e) {
        status.setRollbackOnly(); // Manual rollback
        return false;
    }
});
```

17. Explain realtime Spring Boot Distributed Transactions with Kafka Example?

Distributed transactions ensure **data consistency** across multiple services/databases, especially in **microservices architecture**. Spring Boot supports **event-driven distributed transactions** using **Kafka** as the messaging broker.

📌 Key Concepts

✅ **Two-Phase Commit (2PC)** – Ensures transaction consistency across multiple databases (Not recommended for performance).
✅ **Saga Pattern** – Splits a transaction into **multiple compensating transactions** for eventual consistency.
✅ **Event-Driven Transactions** – Uses **Kafka** to manage transactional events across services asynchronously.

🔹 Example: Distributed Transaction in Order Processing

Scenario

1. **Order Service** creates an order and sends an event (`ORDER_CREATED`) to Kafka.
2. **Payment Service** listens to the event, processes payment, and publishes `PAYMENT_COMPLETED`.
3. **Inventory Service** listens to the event, reserves stock, and commits changes only if payment succeeds.

✅ If any step fails, a **compensating transaction** (rollback) occurs.

Step 1: Configure Kafka Producer in Order Service

Java

```java
@Service
public class OrderService {
    private final KafkaTemplate<String, String> kafkaTemplate;

    public OrderService(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void createOrder(Order order) {
        // Save order in database
        System.out.println("Order Created: " + order.getId());

        // Send event to Kafka
        kafkaTemplate.send("order-topic", "ORDER_CREATED:" + order.getId()
    }
}
```

Step 2: Consume Kafka Event in Payment Service

Java

```java
    @KafkaListener(topics = "order-topic", groupId = "payment-group")
    public void processPayment(String message) {
        System.out.println("Processing Payment for: " + message);

        // Simulating payment success
        boolean paymentSuccess = true;

        if (paymentSuccess) {
            System.out.println("Payment Completed: " + message);
            // Publish event for inventory update
        } else {
            System.out.println("Payment Failed! Rolling back ... ");
        }
    }
}
```

Step 3: Inventory Service Handling Compensation

Java

```java
import org.springframework.stereotype.Service;

@Service
public class InventoryService {

    @KafkaListener(topics = "payment-topic", groupId = "inventory-group")
    public void reserveStock(String message) {
        System.out.println("Reserving Stock for: " + message);

        if (message.contains("PAYMENT_COMPLETED")) {
            System.out.println("Stock Reserved Successfully!");
        } else {
            System.out.println("Stock Reservation Failed! Rolling back ... "
        }
    }
}
```

🔹 Key Benefits of Kafka for Distributed Transactions

✅ **Decouples Microservices** – Services operate independently through Kafka events.

✅ **Supports Eventual Consistency** – Failsafe mechanisms allow retries and rollbacks.

✅ **Improves Performance** – Eliminates blocking transactions across multiple databases.

✅ **Enhances Scalability** – Easily handles high-load transaction processing.

16. Error handling strategies or event replay mechanisms in distributed transactions?

Distributed transactions require robust error handling and event replay mechanisms to ensure data consistency and fault tolerance in microservices. Here are key techniques:

1️⃣ Error Handling Strategies

Distributed systems often face **network failures, service downtime, and inconsistent states**. Effective error handling ensures reliability.

✅ **Retry Mechanism** → Automatically retries failed transactions using **exponential backoff** to prevent overload.
✅ **Dead Letter Queue (DLQ)** → Stores failed events in a **special Kafka queue** for later inspection and processing.
✅ **Idempotency Keys** → Prevents duplicate transaction processing **by ensuring requests are uniquely identifiable**.
✅ **Compensating Transactions (Rollback Actions)** → Undo operations via **Saga Pattern** when failures occur in multi-step transactions.

💡 **Example: Retrying Failed Transactions**

Java Copy

```java
@KafkaListener(topics = "order-topic", groupId = "payment-group")
public void processPayment(String message) {
    try {
        // Process payment
    } catch (Exception e) {
        retryTransaction(message); // Retry logic with backoff strategy
    }
}
```

2️⃣ Event Replay Mechanisms

Event replay allows **recovering lost or failed transactions** by **replaying Kafka events** or **reprocessing database logs**.

✅ **Kafka Consumer Offsets** → Replay events from a specific offset using **consumer group checkpoints**.
✅ **Event Sourcing** → Maintain an **event log** to reconstruct past transactions and restore system state.
✅ **Outbox Pattern** → Stores events **in the database** and later publishes them reliably to Kafka for replay.
✅ **Replay Dead Letter Events** → Periodically process **failed events** from DLQ and reinject them into the system.

💡 **Example: Replaying Kafka Events**

```java
ConsumerRecords<String, String> records = kafkaConsumer.poll(Duration.ofSeconds(5));
for (ConsumerRecord<String, String> record : records) {
    processEvent(record.value()); // Replay stored transactions
}
```